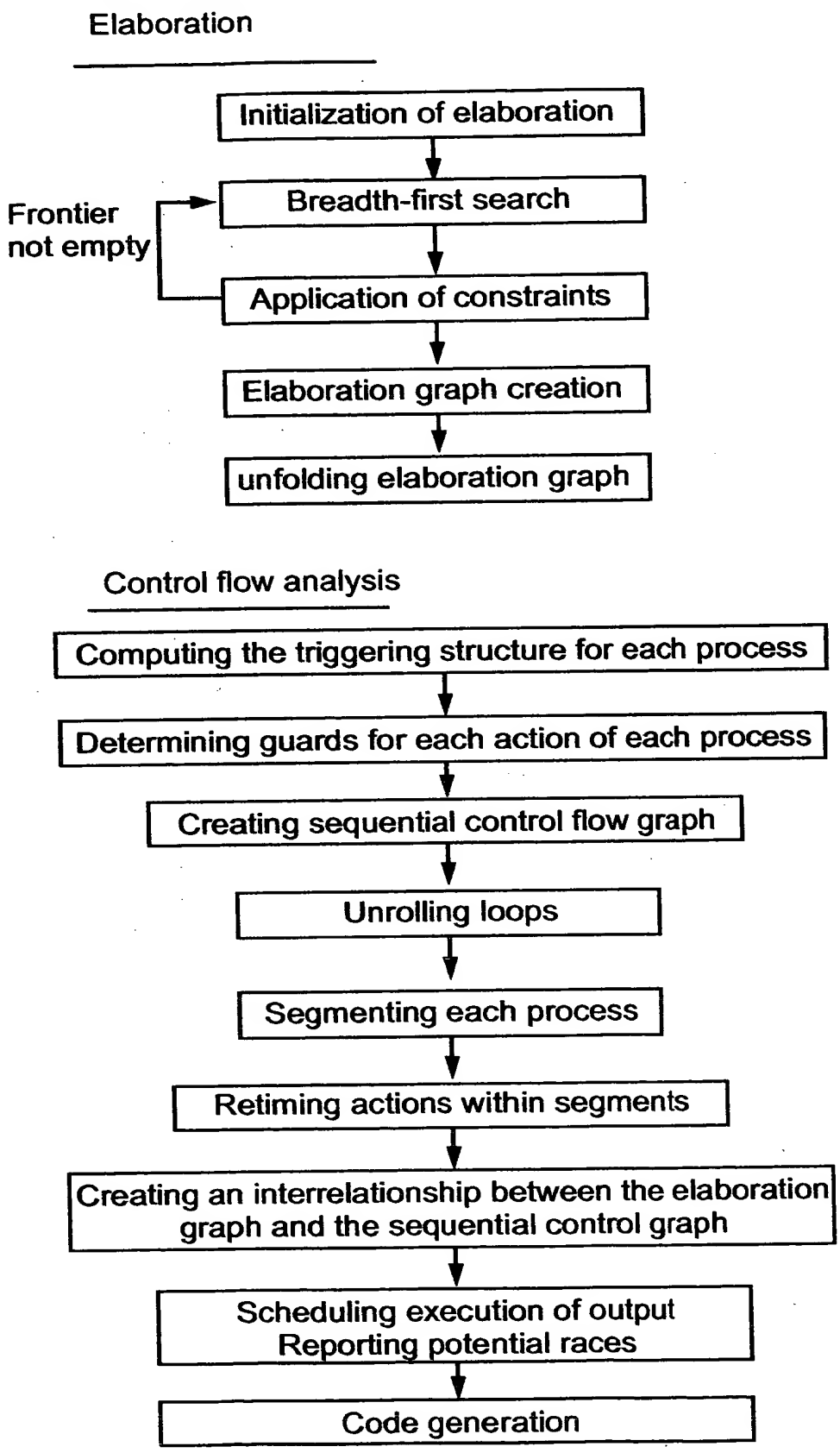

Fig. 1

```
L ← refto( top unit )
C = Φ
while L !=Φ
   for each reference r in L begin
       NL ← Φ
       t ← typeof( r )
       r.target ← makenode( t )
       C ← C+{ constraints of t }
       for each field f in t begin
           NL ← NL + refto( f )
           C ← apply( C )
       end
   end
   L ← L + NL
end
```

Fig. 2

```
<'

struct  cl {                          //  arbiter client
    id              :int;              // my id
    data            :int;              // data   -  INPUT
    !drdy           :bool;             // data ready  -  INPUT
    !xreq           :bool;             // transfer request - interface to arb
    !xgrt           :bool;             // transfer grant - arb  sets this
    arb             :arb;
    keep arb == sys.arb;
};

struct arb {
    cls             :list of cl;
    data            :int ;             // data destination
};

extend sys {
     cl_list        :list  of cl;
     keep cl_list.size () == 4;
     keep for each in cl_list {
          .id == index;
     };
     arb            :arb;
     keep arb.cls == cl_list;
};

'>
```

Fig. 3

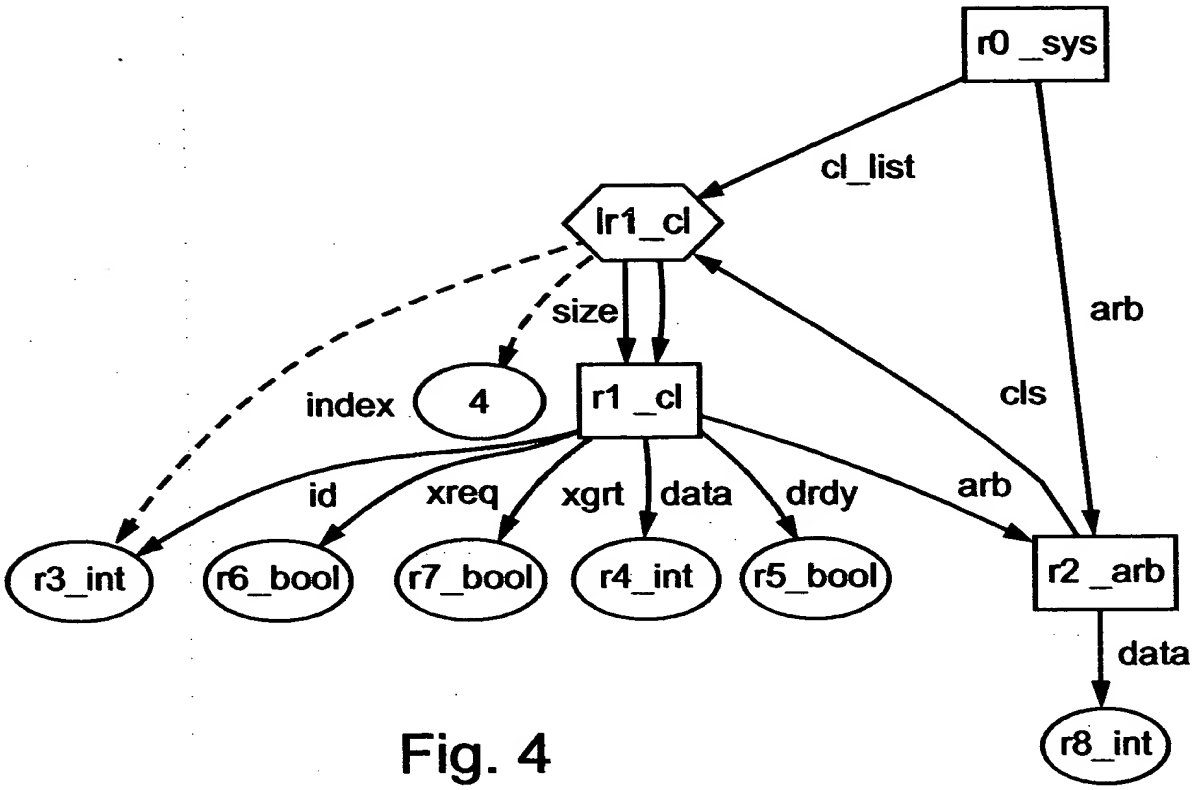

Fig. 4

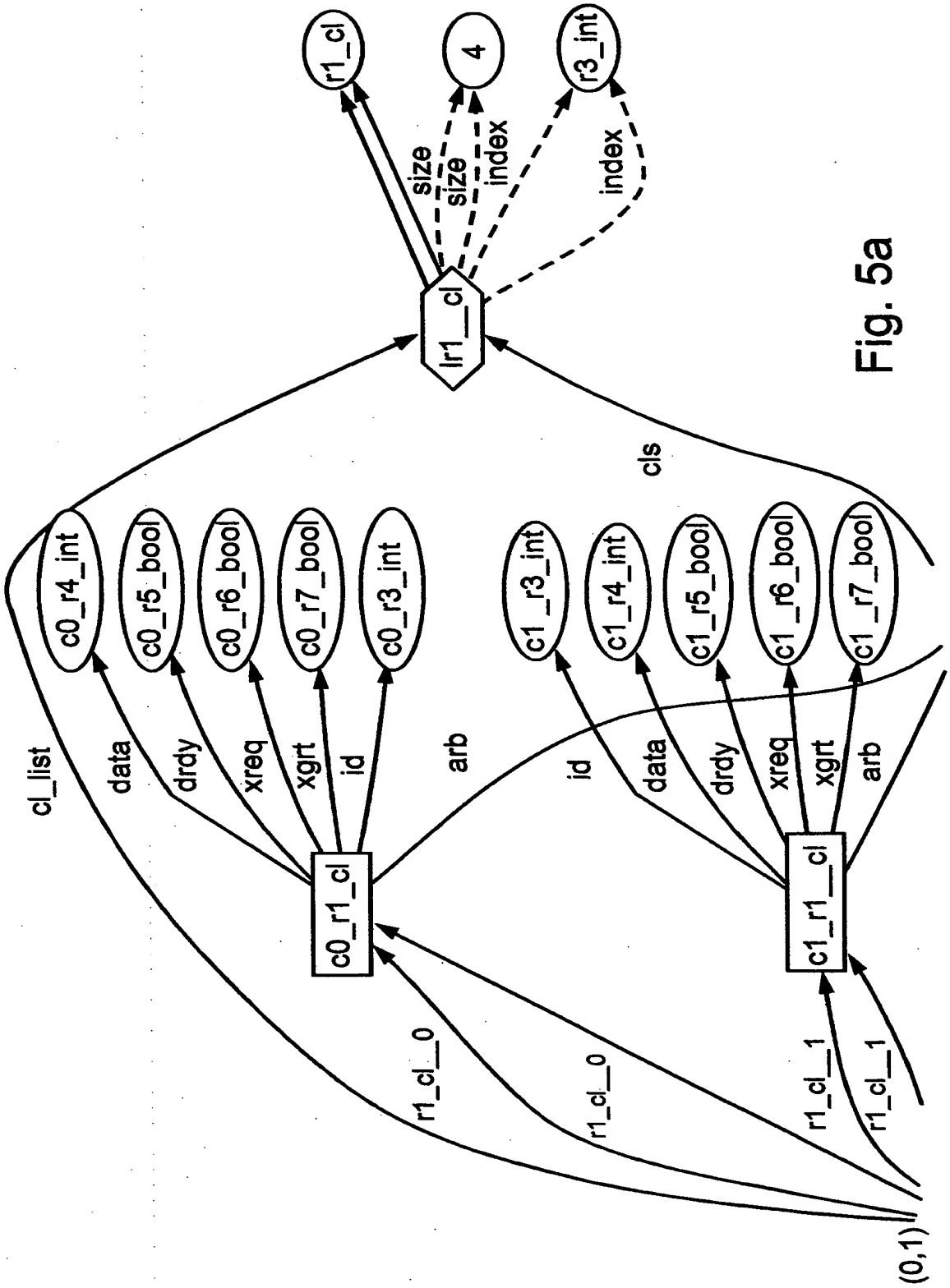

Fig. 5a

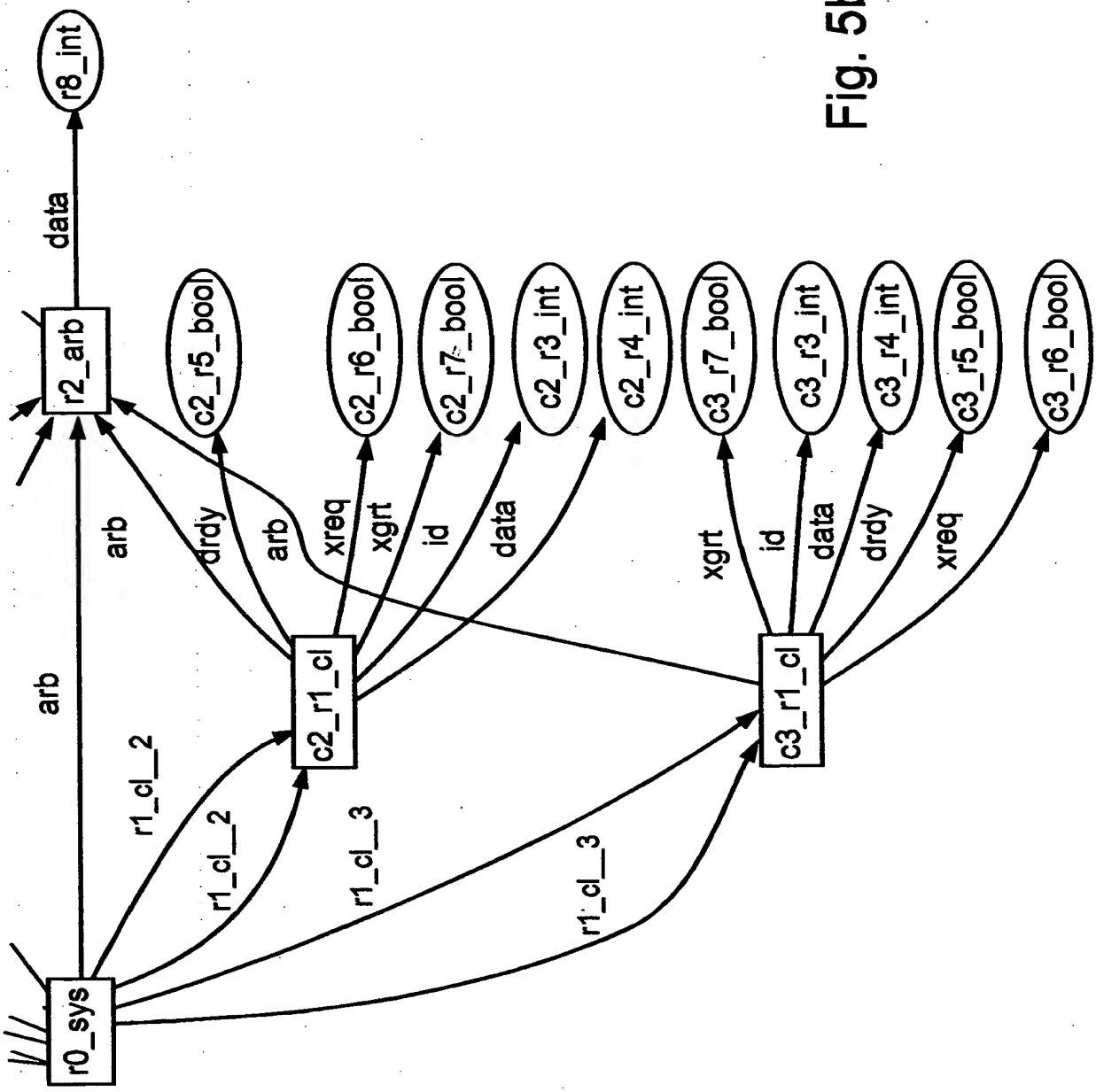

Fig. 5b

```
1   <'
2   struct cl {                                  // arbiter client
3       id          :int;                        // my id
4       data        :int;                        // data - INPUT
5       !drdy        :bool;                       // data ready - INPUT
6       !xreq        :bool;                       // transfer request
7       !xgrt        :bool;                       // transfer grant
8       arb         :arb;
9       keep arb == sys.arb;
10
11      trans() @sys.clk is {
12          while TRUE {
13              wait true(drdy);
14              xreq = TRUE;
15              wait true(xgrt);
16              arb.data = data;
17              wait cycle;
18              xreq = FALSE;
19              wait true(not xgrt);
20              drdy = FALSE;
21          };
22      };
23  };
24
25  struct arb {
26      cls         :list of cl;
27      data        :int;                        // data destination
28      switch() @sys.clk is {
29          while TRUE {
30              for each in cls {
31                  if .xreq then {
32                      .xgrt = TRUE;
33                      wait true(not .xreq);
34                      .xgrt = FALSE;
35                  };
36              };
37              wait cycle;
38          };
39      };
40  };
41
42  extend sys {
43      cl_list     :list of cl;
44      keep cl_list.size() == 4;
45      keep for each in cl_list {
46          .id == index;
47      };
48      arb         :arb;
49      keep arb.cls == cl_list;
50      event clk;
51  };
52  '>
```

Fig. 6

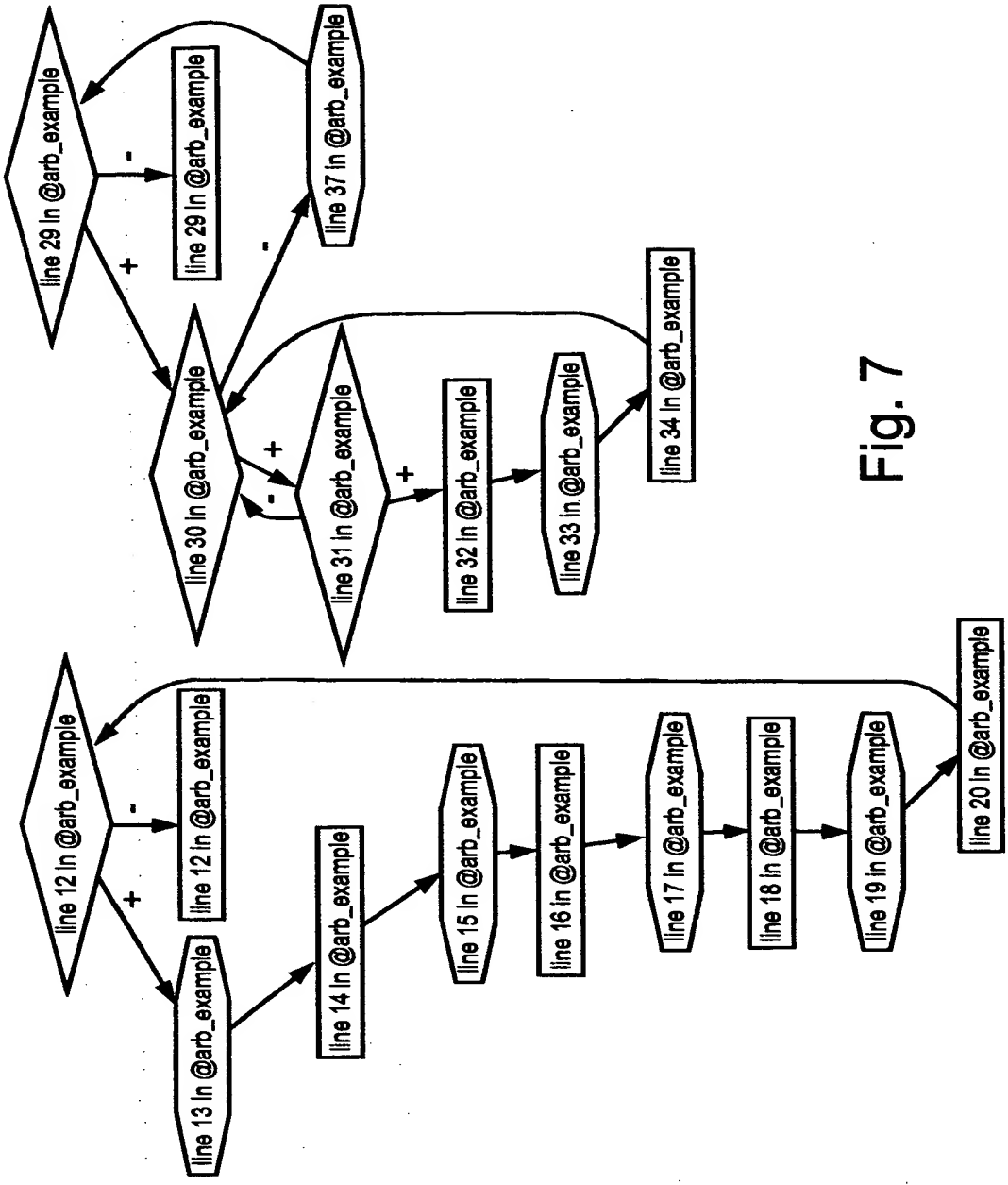

Fig. 7

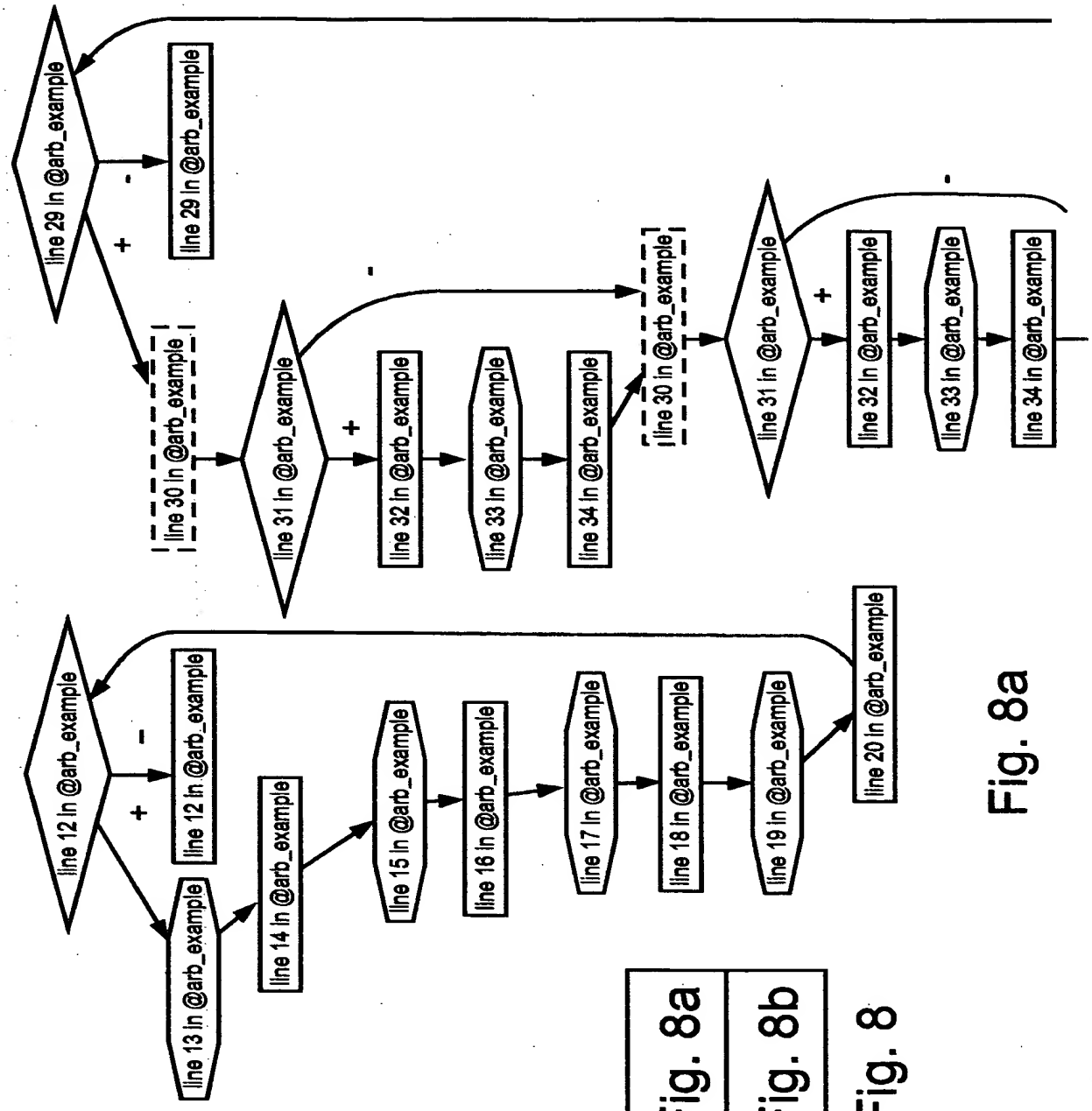

Fig. 8a

Fig. 8b

Fig. 8

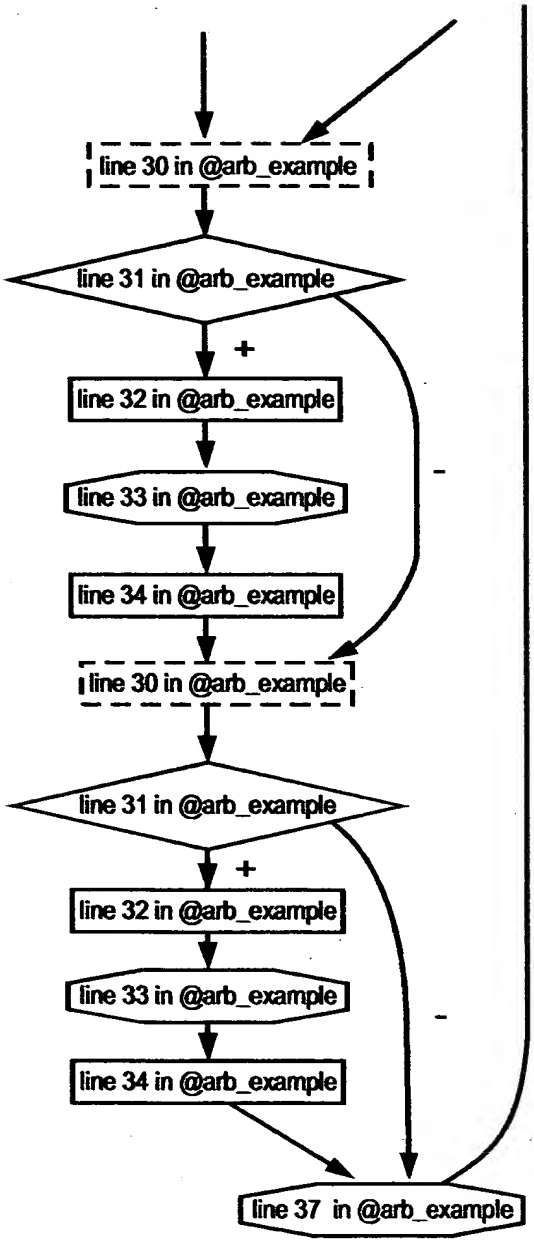

Fig. 8b

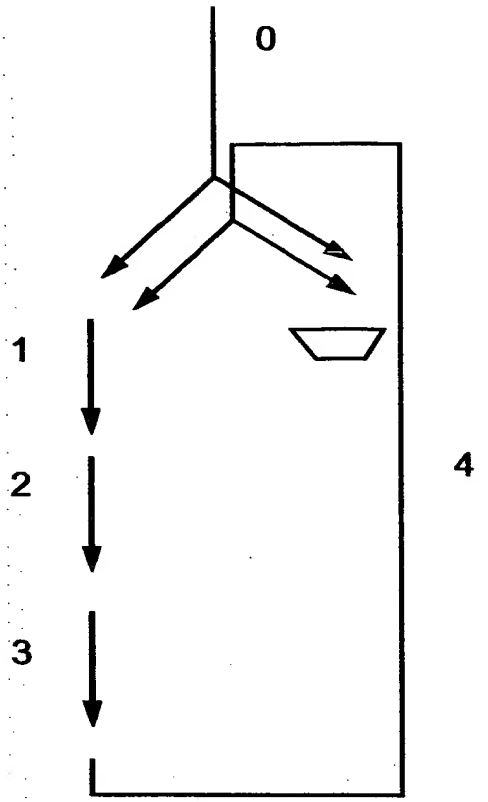

Fig. 9

```
for each node n in EG such that n has processes begin
 for each process p in n begin
  for each segment s in p begin
   for each action a in s begin
    for each read expression e in a begin
     t ← evaluate( e, context )
     tag t with {n, s, 'read'}
    end
    for each write expression e in a begin
     t ← evaluate( e, context )
     tag t with {n, s, 'write'}
    end
   end
  end
 end
end
```

Fig. 10

```
1     Peterson's mutex algorithm - simple two agent example
2
3     <'
4
5     struct agent {
6         req     :bool;
7         id      :int;
8         oa      :agent;
9         p() @sys.clk is {
10            req = TRUE;
11            sys.k = id;
12            while (sys.k == id) && oa.req {
13                wait cycle;
14            };
15            wait cycle;
16            sys.w = id;     // Critical segment
17            req = FALSE;
18        };
19    };
20
21    extend sys {
22        k   :int;   // Requestors id.
23        w   :int;   // The protected data field
24        a0  :agent;
25        a1  :agent;
26        keep a0.id == 0;
27        keep a0.oa == a1;
28        keep a1.id == 1;
29        keep a1.oa == a0;
30        event clk;
31    };
32
33    '>
```

Fig. 11

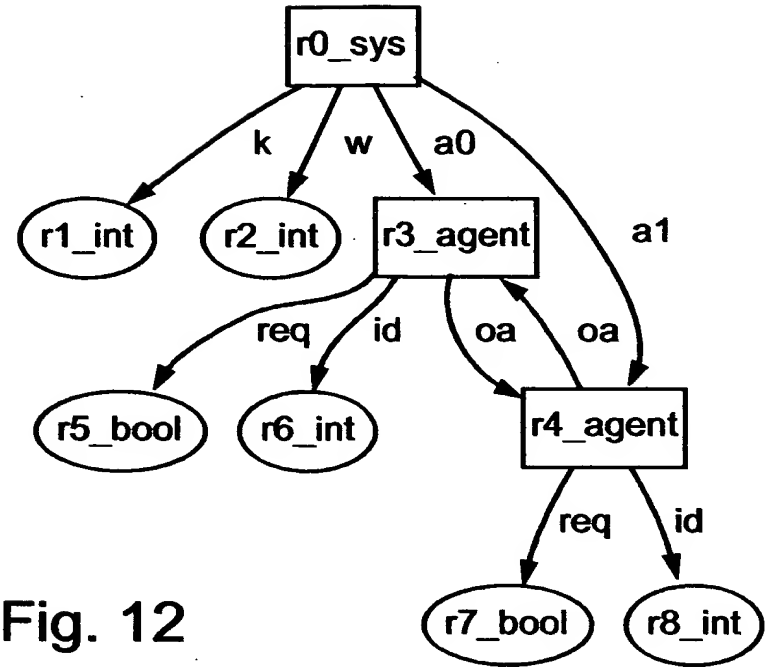

Fig. 12

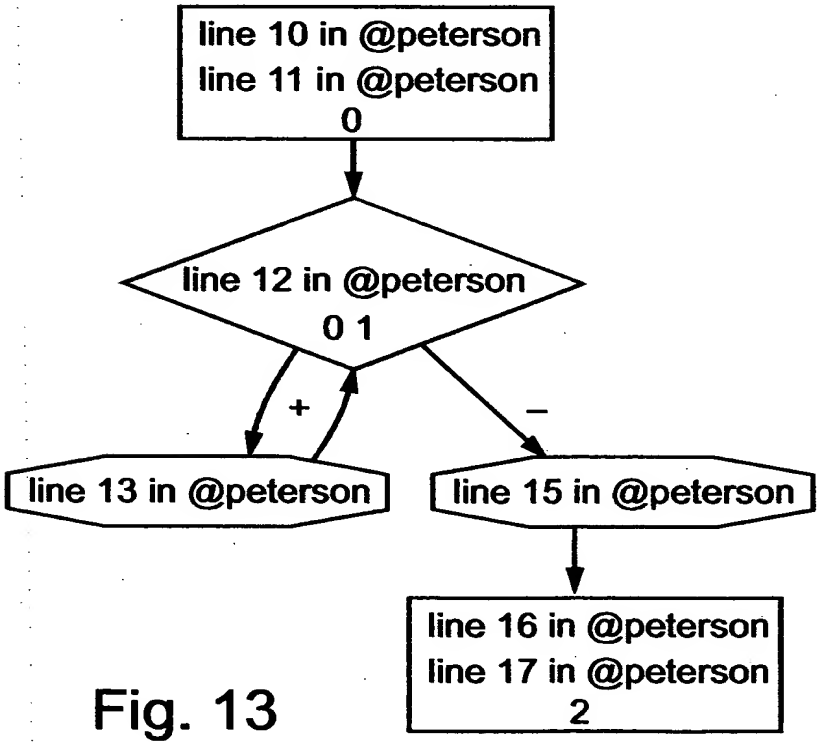

Fig. 13

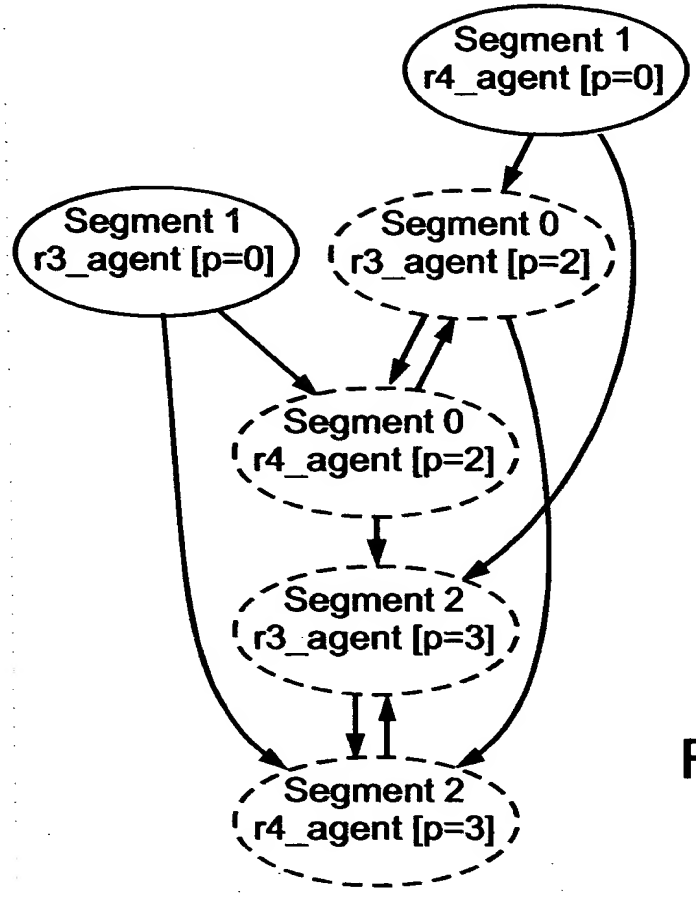

Fig. 14

```
1
2       This is an iterator access to hierarchical arrays
3
4       <'
5       struct ball {
6           dat     :uint (bits:3);
7           mat     :list of bool;
8           keep mat.size() == 2;
9       };
10
11      struct box {
12          flag    :bool;
13          bl     :list of ball;
14          keep bl.size() == 5;
15      };
16
17      struct iter_type {
18          ar      :list of box;
19          foo() @sys.clk is {
20              wait cycle;
21              for each in ar {
22                  .flag = TRUE;
23                  for each in .bl {
24                      .dat = 2;
25                      .mat[1] = FALSE;
26                  };
27              };
28              ar[2].bl[3].mat[0] = TRUE;
29              ar[2].bl[sys.ind].mat[0] = TRUE;
30          };
31      };
32
33      extend sys {
34          event clk;
35          arr     :list of box;
36          keep arr.size() == 4;
37          ind     :int;
38
39          iter    :iter_type;
40          keep iter.ar == arr;
41      };
42
43      '>
```

Fig. 15

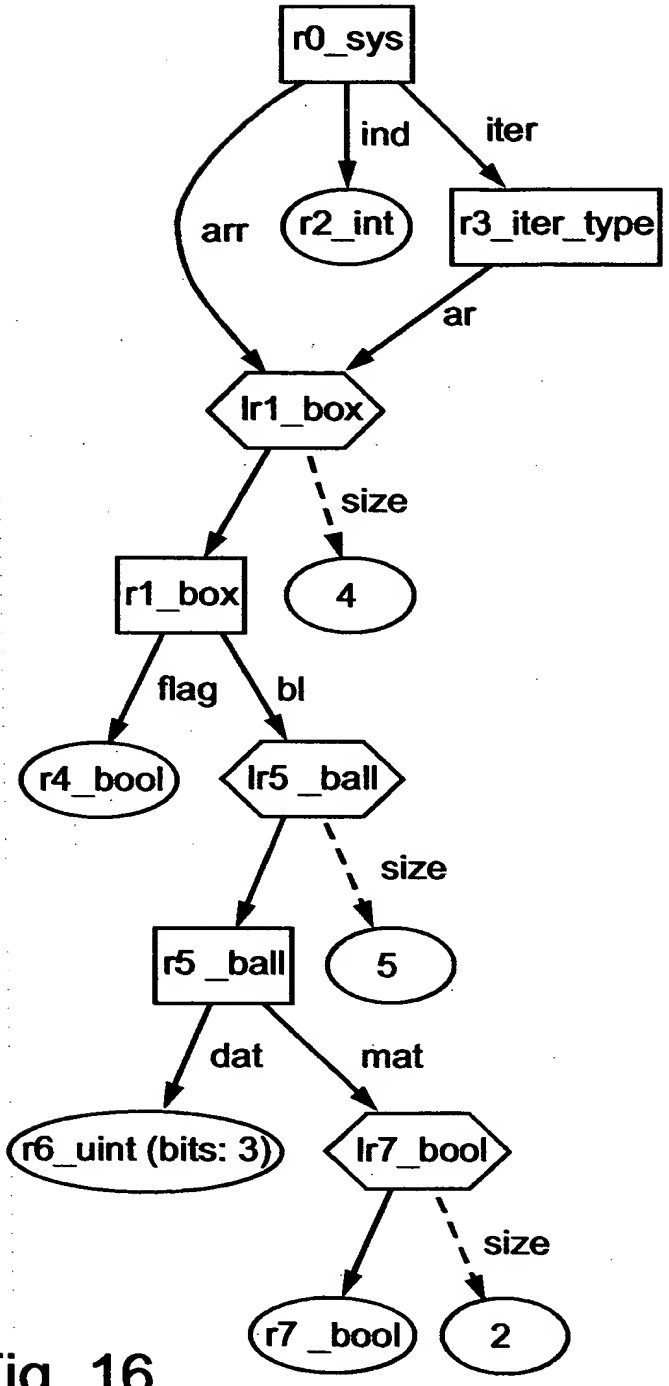

Fig. 16

Figure 1

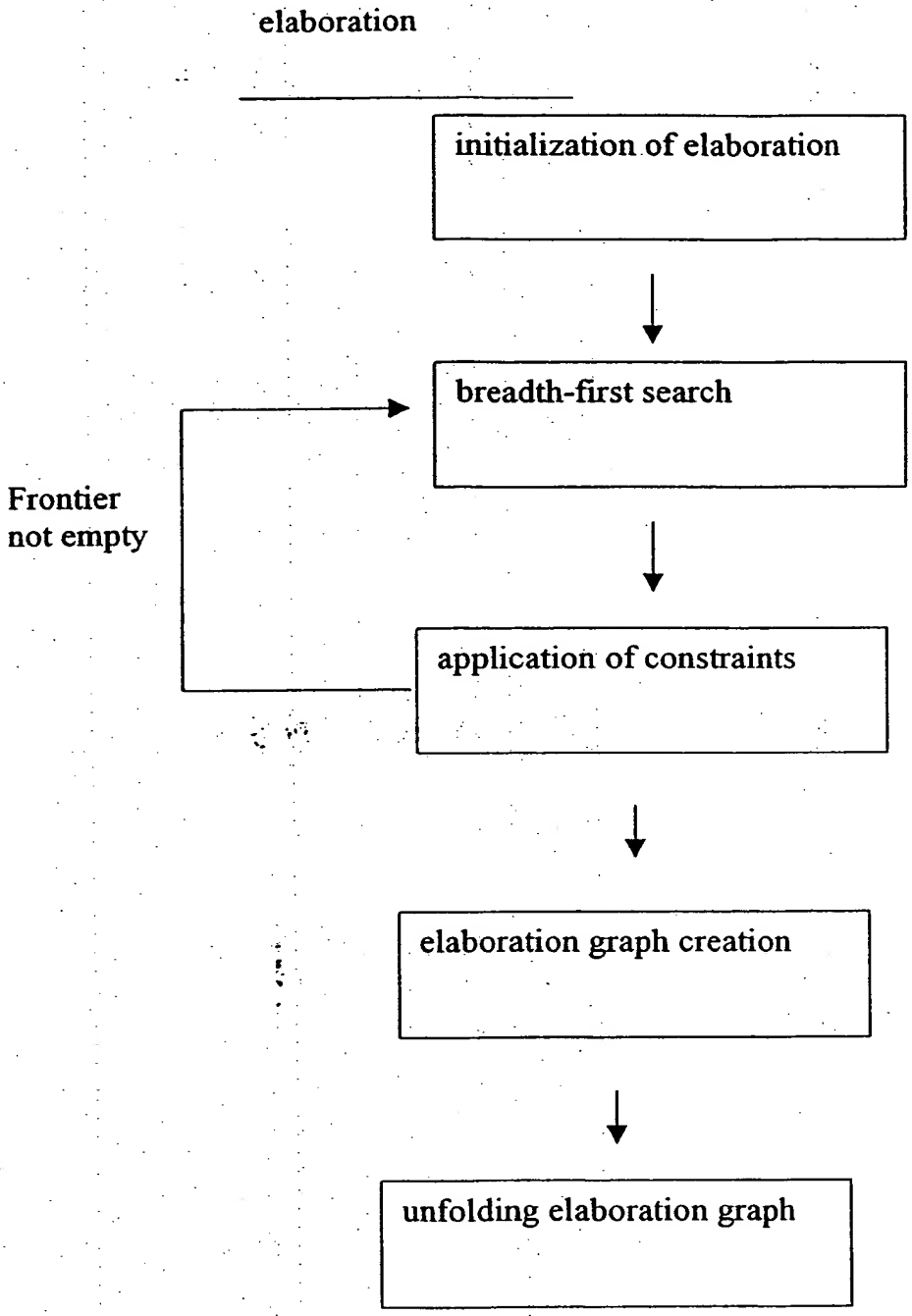

Fig. 1

Figure 1 (con't)

control flow analysis

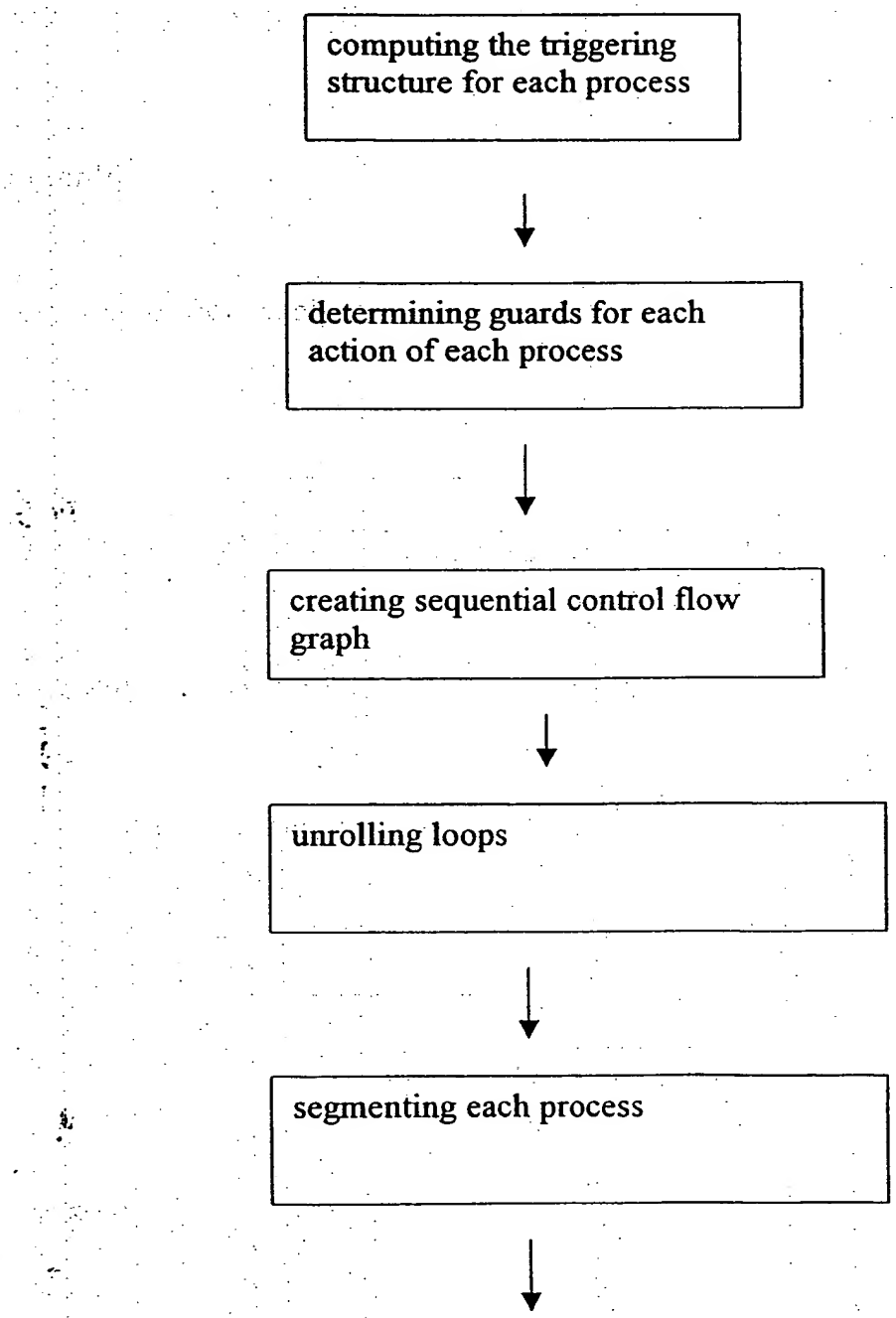

Fig. 1

Figure 1 (con't2)

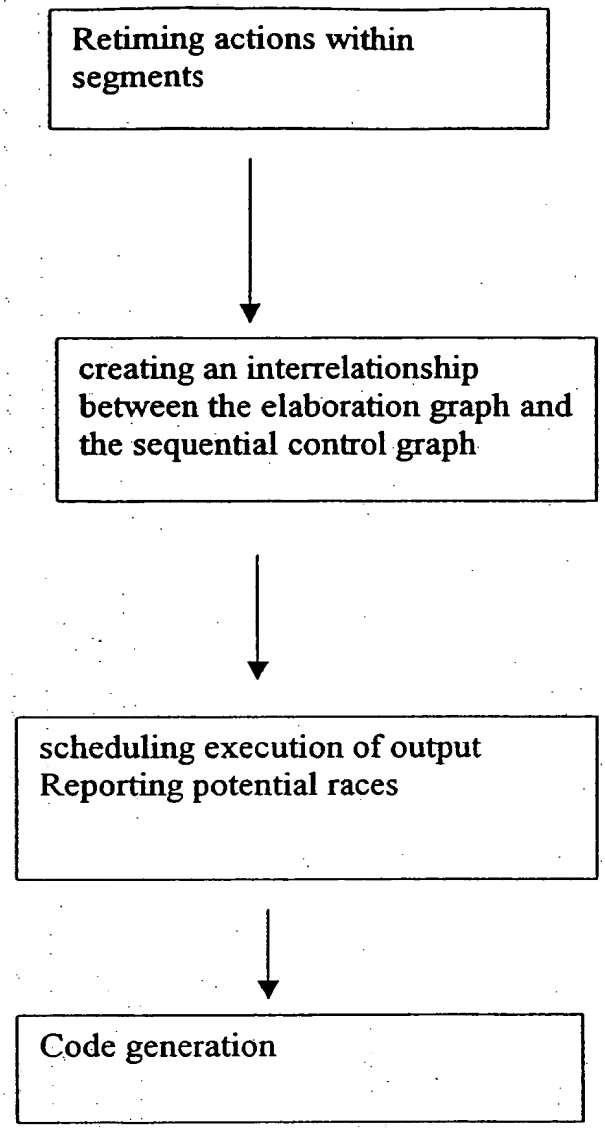

Fig. 1

Figure 8 Part I

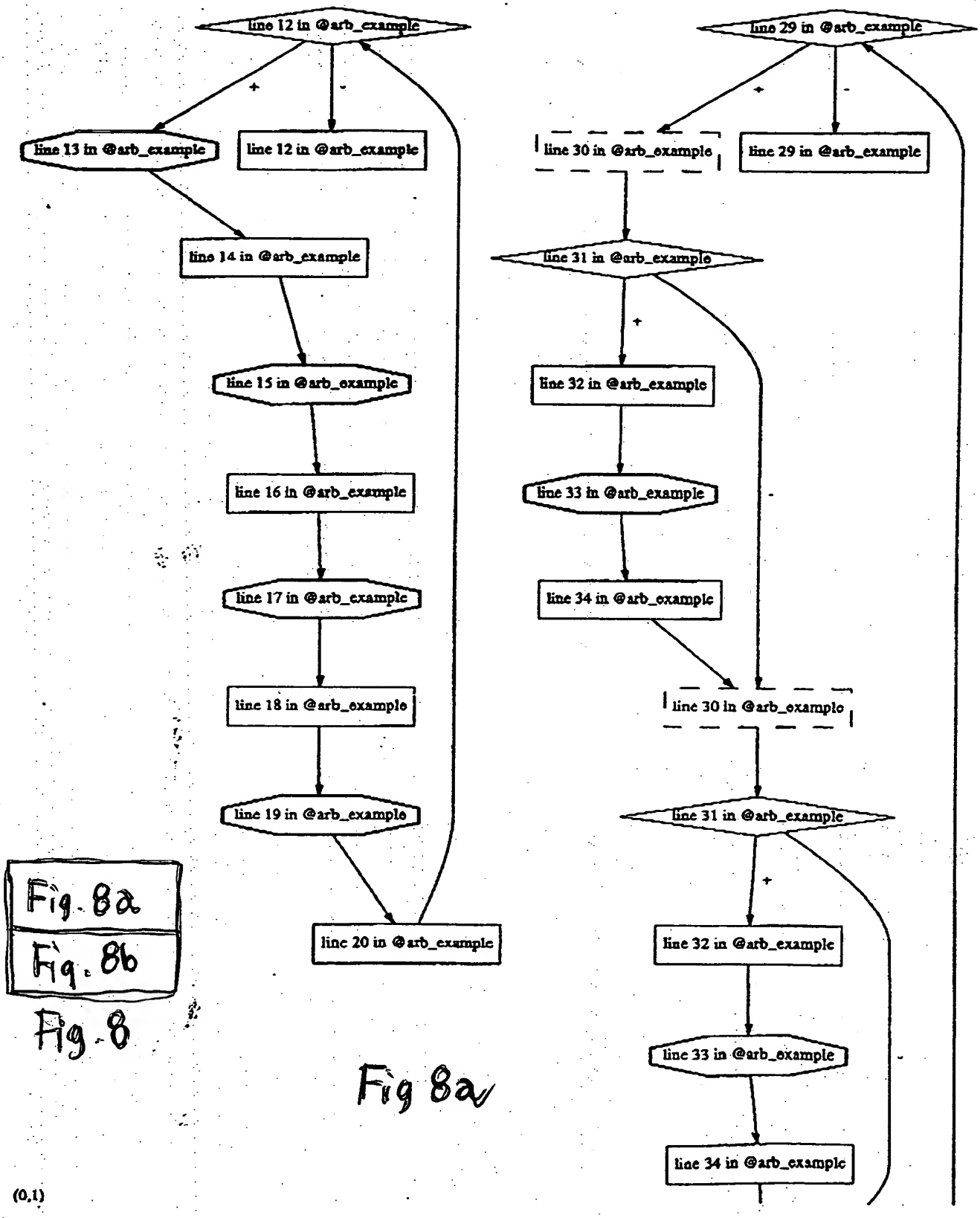

Fig. 8a
Fig. 8b
Fig. 8

Fig 8a

Figure 8 Part II

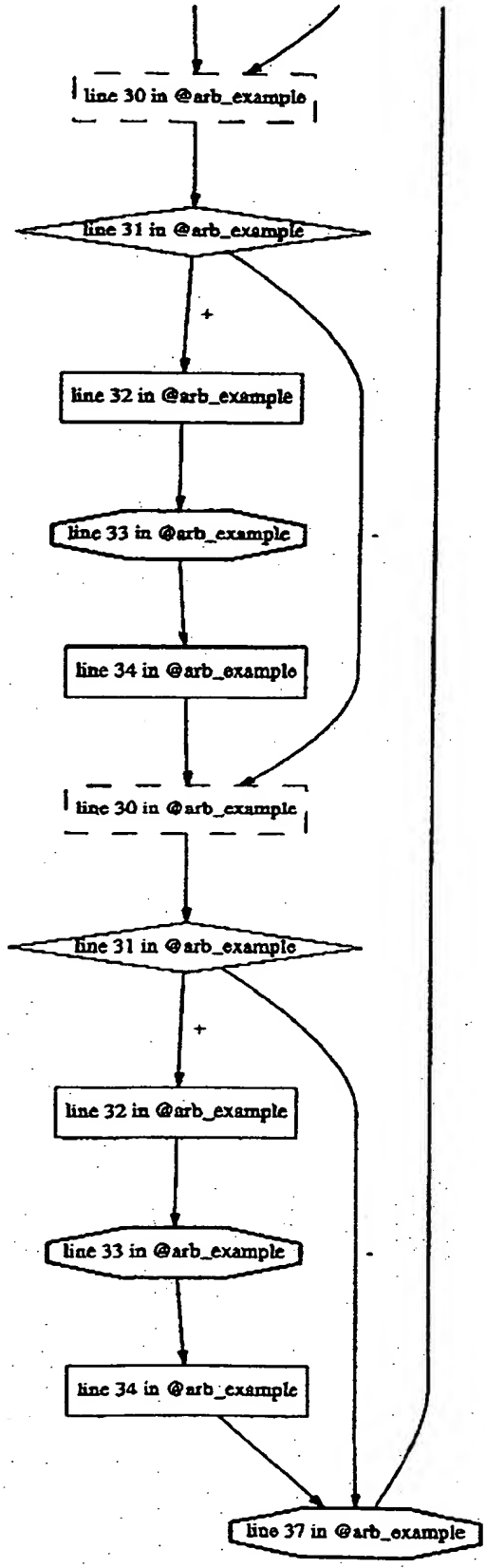

Fig. 8b

(0,0)

Figure 9: Segmentation of a control flow graph

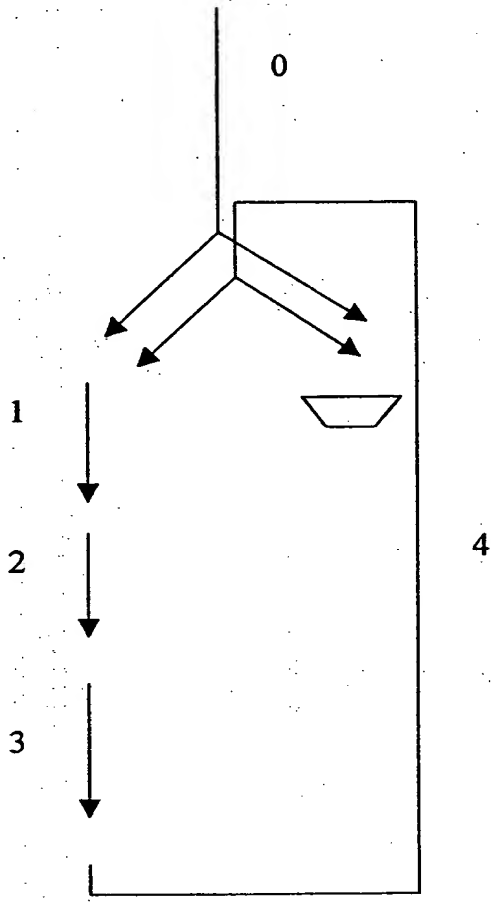

Fig. 9